Gizmosnack

Monday, April 8, 2013

Tutorial - nRF24L01 and AVR

To start out with, you have to know that I am just a hobby programmer so if you find errors or possible improvements in my code, please give me a note so that I can correct them.

If you have read my last blog posts IR-RF remote control and Temperature based flow regulator you have noticed i like to add wireless control to my components. In this tutorial i will describe how i managed to get the nRF24L01 module to work with AVR microships like the Atmega88 (28pin), ATtiny26 (20pin) and ATtiny85 (8pin), since almost all of the tutorials out there are aimed at the Arduino users.

The nRF24L01 module is an awesome RF module that works on the 2,4 GHz band and is perfect for wireless communication in a house because it will penetrate even thick concrete walls. The nRF24L01 does all the hard programming fore you, and even has a function to automatically check if the transmitted data is received at the other end.
There are a couple of different versions of the nRF-family chips and they all seem to work in a similar way. I have for example used the nRF905 (433MHz) module with allmost the same code as I use on the nRF24L01 and the nRF24L01+ without any problems. These little modules has an impressive range, with some versions that manages up to 1000 m (free sight) communication and up to 2000 m with a biquad antenna.

nRF24L01 versus nRF24L01+

The (+) version is the new updated version of the chip and supports data rate of 1 Mbps, 2 Mbps and a "long distance mode" of 250 kbps which is very useful when you want to extend the broadcast length.
The older nRF24L01 (which i have used in my previous posts) only support 1 Mbps or 2 Mbps data rate.
Both the models are compatible with each other, as long as they are set to the same data rate. Since they both costs about the same (close to nothing) I would recommend you to buy the + version!

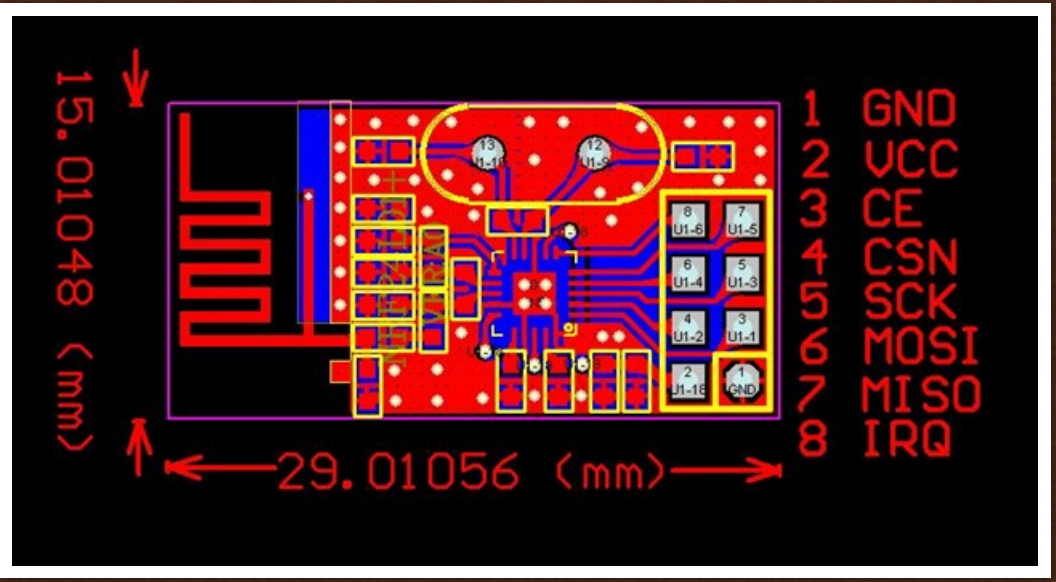
The module

An ebay search on "nRF24L01" shows that there are many different versions of the modules that has the nRF24L01(+), and I have read that some of them are better then others due to better grounding and so on. But if you are after the long-range ones, make sure it has the + sign, and buy one with an extended antenna like this one:

Part one - Setup

Connection differences

The nRF24L01 module has 10 connectors and the + version has 8. The difference is that the + version instead of having two 3,3 V and two GND, have its ground (the one with a white square around it) and 3,3 V supply, next to each other. If changing module from a new + version to an old one, make sure not to forget to move the GND cable to the right place, otherwise it will shorten out your circuit.



Here is a picture of the + version (top view), where you can see all the connections labeled. The old version has two GND connections at the very top instead of at the down right corner.

Power supply (GND & VCC)
The module has to be powered with 3,3 V and **cannot** be powered by a 5 V power supply! Since it takes very little current I use a linear regulator to drop the voltage down to 3,3 V.
To make things a little easier for us, the chip can handle 5 V on the i/O ports, which is nice since it would be a pain to regulate down all the i/O cables from the AVR chip.

Chip Enable (CE)
Is used when to either send the data (transmitter) or start receive data (receiver).
The CE-pin is connected to any unused i/O port on the AVR and is set as output (set bit to one in the DDx register where x is the port letter.)
Atmega88: PB1, ATtiny26: PA0, ATtiny85: PB3

SPI Chip Select (CSN)
Also known as "Ship select not". The CSN-pin is also connected to any unused i/O port on the AVR and set to output. The CSN pin is held high at all the time except for when to send a SPI-command from the AVR to the nRF.
Atmega88: PB2, ATtiny26: PA1, ATtiny85: PB4

SPI Clock (SCK)
This is the serial clock. The SCK connects to the SCK-pin on the AVR.
Atmega88: PB5, ATtiny26: PB2, ATtiny85: PB2

SPI Master output Slave input (MOSI or MO)
This is the data line in the SPI system.
If your AVR chip supports SPI-transfere like the Atmega88, this connects to MOSI on the AVR as well and is set as output.

On AVR's that lacks SPI, like the ATtiny26 and ATtiny85 they come with USI instead, and the datasheet it says:

"The USI Three-wire mode is compliant to the Serial Peripheral Interface (SPI) mode 0 and 1, but
does not have the slave select (SS) pin functionality. However, this feature can be implemented
in software if necessary"

The "SS" refered to is the same as "CSN"
And after some research i found this blog that helped me allot.

To get the USI to SPI up and running I found out that I had to connect the **MOSI** pin from the nRF to the **MISO** pin on the AVR and set it as output.
Atmega88: PB3, ATtiny26: PB1, ATtiny85: PB1

SPI Master input Slave output (MISO or MI)
This is the data line in the SPI system.

If your AVR chip supports SPI-transfere like the Atmega88, this connects to MISO on the AVR and this one stays as an input.

To get it working on the ATtiny26 and ATtiny85, i had to use USI as mentioned above. This only worked when I connected the **MISO** pin on the nRF to the **MOSI** pin on the AVR and set it as input and enable internal pullup.
Atmega88: PB4, ATtiny26: PB0, ATtiny85: PB0

Interrupt Request (IRQ)
The IRQ pin is not necessary, but a great way of knowing when something has happened to the nRF. you can for example tell the nRF to set set the IRQ high when a package is received, or when a successful transmission is completed. Very useful!

If your AVR has more than 8 pins and an available interrupt-pin i would highly suggest you to connect the IRQ to that one and setup an interrupt request.
Atmega88: PD2, ATtiny26: PB6, ATtiny85: -

Part two- Programming
Here i will explain the c-program that runs on the AVR-chip. You can find a working copy of my code here (easier to copy and paste from).

Includes
I have included these lines to get my code to work:

```
#include <avr/io.h>
#include <stdio.h>
#define F_CPU 1000000UL  // 1 MHz change to match your AVR
#include <util/delay.h>
#include <avr/interrupt.h>

#include "nRF24L01.h"
```

You can see that i am importing a file called nRF24L01.h. This is a small library that defines the registers of the nRF so that i for example can call register "STATUS" instead of the register "0x07"... Just copy the text in the link and paste it into a file that you name "nRF24L01.h" and put it in the root of your folder.

Defines
To make the code cleaner i also put these definitions in the "nRF24L01.h"-file:

```
#define BIT(x) (1 << (x))
#define SETBITS(x,y) ((x) |= (y))
#define CLEARBITS(x,y) ((x) &= (~(y)))
#define SETBIT(x,y) SETBITS((x), (BIT((y))))
#define CLEARBIT(x,y) CLEARBITS((x), (BIT((y))))
```

And also add the defines:
`#define W 1`

```
#define R  0
```

SPI

Initialization

The nRF chip communicates with the AVR-chip using SPI which has to be initialized in the AVR according to its datasheet. Here is the initializing code for Atmega88:

```
void InitSPI(void)
{
    //Set SCK (PB5), MOSI (PB3) , CSN (SS & PB2) & CE  as outport
    //OBS!!! Has to be set before SPI-Enable below
    DDRB |= (1<<DDB5) | (1<<DDB3) | (1<<DDB2) |(1<<DDB1);

    // SPI Enable, Master, set clock rate fck/16. clock rate not to important..
    SPCR |= (1<<SPE)|(1<<MSTR);

    SETBIT(PORTB, 2);   //CSN IR_High to start with, nothing to be sent to the nRF yet!
    CLEARBIT(PORTB, 1); //CE low to start with, nothing to send/receive yet!
}
```

ATtiny26:

```
void InitSPI(void)
{
    //Set SCK (PB2), MISO (PB1 connects to nRF MOSI), CSN (PA1) & CE (PA0) as outport
    //OBS!!! Has to be set before SPI-Enable below
    DDRB |= (1<<PB2) | (1<<PB1);
    DDRA |= (1<<PA1) | (1<<PA0);

    ////Set MOSI (PB0) as input OBS: connects to nRF MISO
    DDRB &= ~(1<<PB0);
    PORTB |= (1<<PB0);

    USICR |= (1<<USIWM0)|(1<<USICS1)|(1<<USICLK);

    SETBIT(PORTA, 1);   //CSN high to start with, nothing to be sent to the nRF yet!
    CLEARBIT(PORTA, 0); //CE low to start with, nothing to send/receive yet!
}
```

ATtiny85:

```
void InitSPI(void)
{

    //Set SCK (PB2), MISO (PB1 connects to nRF MOSI) , CSN (PB4) and CE (PB3) as outport
    //OBS!!! Has to be set before SPI-Enable below
    DDRB |= (1<<PB2) | (1<<PB1) | (1<<PB4)| (1<<PB3);

    ////Set MOSI (PB0) as input OBS: connects to nRF MISO
    DDRB &= ~(1<<PB0);
    PORTB |= (1<<PB0);

    USICR |= (1<<USIWM0)|(1<<USICS1)|(1<<USICLK);

    SETBIT(PORTB, 4);   //CSN high to start with, nothing to be sent to the nRF yet!
    CLEARBIT(PORTB, 3); //CE low to start with, nothing to send/receive yet!
}
```

Communication

Now to send and receive a byte from the nRF with the SPI all you have to do is to use this function:

Atmega88 (SPI):

```
char WriteByteSPI(unsigned char cData)
{
    //Load byte to Data register
    SPDR = cData;

    /* Wait for transmission complete */
    while(!(SPSR & (1<<SPIF)));

    //Return what's receaved from the nrf
    return SPDR;
}
```

ATtiny(26 & 85) (USI as SPI):

```
uint8_t WriteByteSPI(uint8_t cData)
{
    //Load byte to Data register
    USIDR = cData;

    USISR |= (1<<USIOIF); // clear flag to be able to receive new data

    /* Wait for transmission complete */
    while ( (USISR & (1<<USIOIF)) == 0 )
    {
        USICR |=(1<<USITC);     //Toggle SCK and count a 4-bit counter from 0-15, when it reaches 15 USIOIF is set!
    }

    return USIDR;
}
```

I don't think it matters if you send a char or an integer, this is just how i got it to work... Note that these functions always returns something, this returned message is only cared fore when reading data from the nRF (a more appropriate name of the function might be "Write_Read_Byte_SPI").

nRF24L01(+) communication

Now to the fun part...

How it works

1) The nRF starts listening for commands when the CSN-pin goes low.
2) after a delay of 10us it accepts a single byte through SPI, which tells the nRF which bytes you want to read/write to, and if you want to read or write to it.
3) a 10us delay later it then accepts further bytes which is either written to the above specified register, or a number of dummy bytes (that tells the nRF how many bytes you want to read out)
4) when finished close the connection by setting CSN to high again.

Reading bytes from nRF

To start off, make sure your SPI communication is working by reading out something from the nRF. Reading a register on the nRF is accomplished by this function: (all of my example codes is for Atmega88, just change to the right port and pin number for the CSN and CE to get it to work on ATtiny as well)

```
uint8_t GetReg(uint8_t reg)
{
    _delay_us(10); //make sure last command was a while ago...
    CLEARBIT(PORTB, 2); //CSN low - nRF starts to listen for command
    _delay_us(10);
    WriteByteSPI(R_REGISTER + reg); //R_Register = set the nRF to reading mode, "reg" = this registry will be read back
    _delay_us(10);
    reg = WriteByteSPI(NOP);    //Send NOP (dummy byte) once to receive back the first byte in the "reg" register
    _delay_us(10);
    SETBIT(PORTB, 2);   //CSN Hi - nRF goes back to doing nothing
    return reg; // Return the read registry
}
```

I recommend you to start out by reading the STATUS register like this:

USART

If you have an AVR that supports USART like the Atmega88, i highly recommend you to use that as a way of sending the data back to the computer with this little friend... (I have written a small tutorial in the subject)

This is done simply by calling the function like this:

```
USART_Transmit(GetReg(STATUS));
```

If you like me have a function called "USART_Transmit(uint8_t data)"
The usart should send 0b00001110 (or 0x0E) to the computer since it is the preset configuration of the STATUS registry (see the end of this blogpost).

LED
If you are using an ATtiny it lacks the USART and thereby the ability to write things back to the computer, then you can use a more hardcore way using an LED to turn on if the STATUS register is set correctly. Since the bites in the STATUS (0x07) register are preset to 0b00001110 (or 0x0E) you can test if this is true by this function:

```
if (GetReg(STATUS)==0x0E)
{
    SETBIT(PORTB, 5); //LED on when true
}
```

Make sure you remember to first set the LED-pin to output: DDRB |=(1<<5);

If you are using an 8-pin ATtiny like the ATtiny85, there is not a single free pin on the chip to put the LED on, so i think a good idea would be to use the MISO-pin as a temporary LED output (since it is an output already and the SPI is not in use at the moment). Attach the LED to the PB1 and via a resistor to GND, then in the if-function above change the port number to 1. I haven't tested this my selves but i doubt it would cause any problem to the SPI-connection.

Writing bytes to the nRF
Now it's time to send a command to the nRF, this is done almost the exact same way as the reading command with this function:

```
void WriteToNrf(uint8_t reg, uint8_t Package)
{
    _delay_us(10); //make sure last command was a while ago...
    CLEARBIT(PORTB, 2); //CSN low - nRF starts to listen for command
    _delay_us(10);
    WriteByteSPI(W_REGISTER + reg); //W_Register = set the nRF to Write mode, "reg" = this registry will be written to
    _delay_us(10);
    WriteByteSPI(Package);  //Send the package to be written to the registry "reg"
    _delay_us(10);
    SETBIT(PORTB, 2);   //CSN Hi - nRF goes back to doing nothing
}
```

If the register holds more then one byte, the TX_ADDR-byte for example holds five bytes, then you have to send them one at a time after each other with 10us delay in between. This makes the function a bit more complicated since C-code is unwilling to pas arrays of integers into functions as is.

I also wanted to clean up my code a bit, so I decided to make one function that I can use to both read and write to the nRF. The function should also accept an array of integers and be able to return an array of integers. This is the result:

```
uint8_t *WriteToNrf(uint8_t ReadWrite, uint8_t reg, uint8_t *val, uint8_t antVal)
{
    //"ReadWrite" ("W" or "R"), "reg" (the register), "*val" (an array with the package) & "antVal" (number of integers in the package)

    if (ReadWrite == W) //if "W" then you want to write to the nRF (read mode "R" == 0x00, so skipping that one)
    {
        reg = W_REGISTER + reg; //Add the "write" bit to the "reg"
    }

    //Create an array to be returned at the end
    //Static uint8_t is needed to be able to return an array (notice the "*" to the left of the function: *WriteToNrf())
    static uint8_t ret[32];

    _delay_us(10);      //make sure last command was a while ago...
    CLEARBIT(PORTB, 2); //CSN low - nRF starts to listen for command
    _delay_us(10);
    WriteByteSPI(reg);  //set the nRF to Write or read mode of "reg"
    _delay_us(10);

    int i;
    for(i=0; i<antVal; i++)
    {
        if (ReadWrite == R && reg != W_TX_PAYLOAD)  //Did you want to read a registry?
        {                                           //When writing to W_TX_Payload you cannot add the "W" since it is on the same level in the registry...
            ret[i]=WriteByteSPI(NOP);   //Send dummy bytes to read out the data
            _delay_us(10);
        }
        else
        {
            WriteByteSPI(val[i]);   //Send the commands to the nRF once at a time
            _delay_us(10);
        }
    }
    SETBIT(PORTB, 2);   //CSN Hi - nRF goes back to doing nothing

    return ret; //return the array
}
```

The most confusing thing whit this function is the W_TX_PAYLOAD in the if-statement... The thing is that when you want to write bytes to the W_TX_PAYLOAD you cannot add the W_REGISTER as you normally does when you want to write to a register. Have a look at the registry setup at the very bottom of this blog post, and you will see that the W_REGISTER and the W_TX_PAYLOAD is in the same "top level" of the registers. The same goes for the TX_FLUSH registers...

Now here is some examples that shows how to use the function:

```
uint8_t val[5]; //An array av integers to send to the *WriteToNrf function

//Write one byte to the nRF:

//EN_RXADDR registry
val[0]=0x01;    //chose which of the data pipes to enable (0-5), now just nr 0.
WriteToNrf(W, EN_RXADDR, val, 1); //enable data pipe 0

//Write 5 bytes to the nRF:

//RX_ADDR_P0 registry. 1-5 bytes - the receiver address in channel P0
int i;
for(i=0; i<5; i++)
{
    val[i]=0x12;    //RF channel 0 address 0x12 x 5 - (make sure to put the same address on the transmitter!!)
}
WriteToNrf(W, RX_ADDR_P0, val, 5);


//Read an array of bytes from the nRF:
uint8_t *data;

data=WriteToNrf(R, RX_ADDR_P0, data, 5);    //store the Receiver address in the "data"-array
```

To come back to the W_TX_PAYLOAD, when you want to ad the payload to the nRF, you simply use the "R" instead of the "W" to trick the WeiteToNrf-function to not add the W_REGISTER.

Setting up nRF24L01(+)

Now it is time to setup the nRF for your specifications. In the example codes, I will send a 5 byte payload with the nRF. This is easily changed to a 1-32 byte payload by changing a bit in the initialization step of the nRF (see below)... This is how i usually set it up for simple communication between two nRF's:

```
void nrf24L01_init(void)
{
    _delay_ms(100); //allow radio to reach power down if shut down

    uint8_t val[5]; //An array av integers to send to the *WriteToNrf function

    //EN_AA - (enable auto-acknowledgments) - Transmitter gets automatic response from receiver when successful transmission! (lovely function!)
    //Only works if Transmitter has identical RF_Adress on its channel ex: RX_ADDR_P0 = TX_ADDR
    val[0]=0x01;    //set value
    WriteToNrf(W, EN_AA, val, 1);   //W=write mode, EN_AA=register to write to, val=data to write, 1=number of data bytes.

    //Choose number of enabled data pipes (1-5)
    val[0]=0x01;
    WriteToNrf(W, EN_RXADDR, val, 1); //enable data pipe 0

    //RF_Adress width setup (how many bytes is the receiver address, the more the merrier 1-5)
    val[0]=0x03;    //0b0000 00011 = 5 bytes RF_Adress
    WriteToNrf(W, SETUP_AW, val, 1);

    //RF channel setup - choose frequency 2,400-2,527GHz 1MHz/step
    val[0]=0x01;    //0b0000 0001 = 2,401GHz (same on TX and RX)
    WriteToNrf(W, RF_CH, val, 1);

    //RF setup  - choose power mode and data speed. Here is the diference with the (+) version!!!
    val[0]=0x07;    //00000111 bit 3="0" 1Mbps=longer range, bit 2-1 power mode ("11" = -0dB ; "00"=-18dB)
    WriteToNrf(W, RF_SETUP, val, 1);

    //RX RF_Adress setup 5 byte - Set Receiver address (set RX_ADDR_P0 = TX_ADDR if EN_AA  is enabled!!!)
    int i;
    for(i=0; i<5; i++)
    {
        val[i]=0x12;    //0x12 x 5 to get a long and secure address.
    }
    WriteToNrf(W, RX_ADDR_P0, val, 5); //since we chose pipe 0 on EN_RXADDR we give this address to that channel.
    //Here you can give different addresses to different channels (if they are enabled in EN_RXADDR) to listen on several different transmitters)

    //TX RF_Adress setup 5 byte -  Set Transmitter address (not used in a receiver but can be set anyway)
    for(i=0; i<5; i++)
    {
        val[i]=0x12;    //0x12 x 5 - same on the Receiver chip and the RX-RF_Address above if EN_AA is enabled!!!
    }
    WriteToNrf(W, TX_ADDR, val, 5);

    //Payload Width Setup - 1-32byte (how many bytes to send per transmission)
    val[0]=5;   //Send 5 bytes per package this time (same on receiver and transmitter)
    WriteToNrf(W, RX_PW_P0, val, 1);

    //CONFIG reg setup - Now it's time to boot up the nrf and choose if it's suppose to be a transmitter or receiver
    val[0]=0x1E;  //0b0001 1110 - bit 0="0":transmitter bit 0="1":Receiver, bit 1="1"=power up,
                  //bit 4="1"= mask_Max_RT i.e. IRQ-interrupt is not triggered if transmission failed.
    WriteToNrf(W, CONFIG, val, 1);

    //device need 1.5ms to reach standby mode (CE=low)
    _delay_ms(100);


}
```

In the code above, i missed a very important setting (if using EN_AA) that sets the number of retries and the retry delay like this:

```
val[0]=0x2F;    //0b0010 00011 "2" sets it up to 750uS delay between every retry (at least 500us at 250kbps and if payload >5bytes in 1Mbps,
                //and if payload >15byte in 2Mbps) "F" is number of retries (1-15, now 15)
WriteToNrf(W, SETUP_RETR, val, 1);
```

Add these lines in the function above to set the number of retries to 15 and the delay to 750us... the delay has to be greater than 500us if you are in the 250kbps mode, or if the payload is greater than 5bytes when in 1Mbps-mode or a payload greater then 15bytes when in 2Mbps mode. Note that the default value of this is only 250us, and will therefore cause trouble when in 250kbps mode and with bigger payloads!

As you can see i decided to make it a transmitter this time. This is easily changed in the code by first delay 50ms, to make sure the nRF is in sleep mode, then send 0x1F to the CONFIG register, than make it wait 50ms again before the first receive-command.

Transmit data

When in transmitting mode this is the function that sends your payload:

```
void transmit_payload(uint8_t * W_buff)
{
    WriteToNrf(R, FLUSH_TX, W_buff, 0); //Sends 0xE1 to flush the registry from old data! W_buff[] is only there because an array has to be called with an array..
    WriteToNrf(R, W_TX_PAYLOAD, W_buff, 5); //Sends the data in W_buff to the nrf
    //Why Flush_TX and W_TX_Payload is sent with an "R" instead of "W" is because they are on the highest byte-level in the nRF (see datasheet below)!

    //sei(); //Enable global interrupt (if interrupt is used)

    _delay_ms(10);      //needs a 10ms delay to work after loading the nrf with the payload for some reason
    SETBIT(PORTB, 1);   //CE high=transmit the data!
    _delay_us(20);      //delay at least 10us!
    CLEARBIT(PORTB, 1); //CE low = stop transmitting
    _delay_ms(10);      //long delay again before proceeding.
}
```

And you call the transmit function like this: (now i just send 0x93 five times in a row, you can fill the array with any bytes you want to send)

```
uint8_t W_buffer[5];
int i;
for (i=0;i<5;i++)
{
    W_buffer[i]=0x93;
}

transmit_payload(W_buffer);
```

Receive data

When in receiver mode this is the function that listens and receives your data:

```
void receive_payload(void)
{
    //sei();        //Enable global interrupt (if interrupt is used)

    SETBIT(PORTB, 1);   //CE IR_High = "listens" for data
    _delay_ms(1000);    //listens for 1s at a time
    CLEARBIT(PORTB, 1); //ce low again -stop listening

    //cli();    //Disable global interrupt
}
```

After every received/transmitted payload the IRQ's in the nRF has to be reset in order to receive/transmit next package. This is done like this:

```
void reset(void)
{
    _delay_us(10);
    CLEARBIT(PORTB, 2); //CSN low
    _delay_us(10);
    WriteByteSPI(W_REGISTER + STATUS);  //write to STATUS registry
    _delay_us(10);
    WriteByteSPI(0x70); //Reset all irq in STATUS registry
    _delay_us(10);
    SETBIT(PORTB, 2);   //CSN IR_High
}
```

Verify transmitted/received data using Interrupt

If you have more than an 8-pin AVR, I say use an interrupt to get triggered when data is successfully received or transmitted. INT0 interrupt is setup like this:

First you have to initialize the interrupt like this on Atmega88:

```
void INT0_interrupt_init(void)
{
    DDRD &= ~(1<<DDD2); //Extern interrupt on INT0, make sure it is input

    EICRA |=  (1<<ISC01);   // INT0 falling edge    PD2
    EICRA  &=  ~(1<<ISC00); // INT0 falling edge    PD2

    EIMSK |=  (1<<INT0);    //enable int0 interrupt
    //sei();              // Enable global interrupts do later
}
```

And initialization on ATtiny26:

```
void INT0_interrupt_init(void)
{
    DDRB &= ~(1<<DDB6); //External interrupt on INT0, make sure it is input

    MCUCR |=  (1<<ISC01);   // INT0 falling edge    PB6
    MCUCR  &=  ~(1<<ISC00); // INT0 falling edge    PB6

    GIMSK |=  (1<<INT0);    //enable int0 interrupt
    //sei();              // Enable global interrupts do later
}
```

Interrupt caused when receiving data

Setup a function triggered by the corresponding vector (INT0) at the very bottom of your code like this: (the global array "*data" is at the very top of my code...) And here i use it to send the received payload to the computer by usart:

```
uint8_t *data;

ISR(INT0_vect)  //vektorn that is run when transmit_payload succeed or when receive_payload received data
{
    //OBS: Mask_Max_rt in config registry has to be set to stop it from trigger on failed transmission!

    cli();  //Disable global interrupt
    CLEARBIT(PORTB, 1);     //ce low -stop sending/listening

    SETBIT(PORTB, 0); //led on to show success
    _delay_ms(500);
    CLEARBIT(PORTB, 0); //led off

    data=WriteToNrf(R, R_RX_PAYLOAD, data, 5);  //read out received message
    reset();    //reset the chip for further comunication

    for (int i=0;i<5;i++)
    {
        USART_Transmit(data[i]);    //send the received data to the computor with usart
    }

    sei();

}
```

Interrupt caused on transmission success
Use the same interrupt function when transmitting data but change its content to just flash the LED to tell you that transmission completed, or you can tell the nRF to switch for a receiver, if what you just sent was a question to a receiver that in turn changed to a transmitter to return your call.

When you use interrupt, it is crucial that you enables the external interrupts by the command sei(); This should be done before the receive_payload, and transmit_payload is used.

Verify transmitted received data without interrupt

If your chip lacks interrupt or if you ran out of free interrupt pins, you can manually check if the IRQ-flags are set in the status register after every time you either transmit data, or run the receive_payload function (before the reset function!!!)

Transmitting
The easiest when you want to see if transmission succeeded or failed, is to check if the MAX_RT is set which mean it failed. This is done like this:

```
if((GetReg(STATUS) & (1 << 4)) != 0 )   //if bit:4 MAX_RT is "1" then transmission failed!
```

Then you know you have to resend the package.... i usually put this statement in a while loop that loops untill the package is received!

Receiving
Do the same check to see if no data is received by checking the RX_DR-bit (data-ready) like this:

```
if (((GetReg(STATUS) & (1 << 6)) != 0 )) //Set high when new data arrives
```

Or it might be a better idea to check if data is received by changing "!=" to "==", if not, you start the receive_payload function again!

Code overview

```c
#include <avr/io.h>
#include <stdio.h>
#define F_CPU 1000000UL  // 1 MHz
#include <util/delay.h>
#include <avr/interrupt.h>

#include "nRF24L01.h"

uint8_t *data;

/****************SPI*****************************/
void InitSPI(void)...
char WriteByteSPI(unsigned char cData)...

/****************in/out LED*************************/
void ioinit(void)           ...

/****************interrupt*************************/
void INT0_interrupt_init(void)  ...

/****************Functions*************************/
uint8_t *WriteToNrf(uint8_t ReadWrite, uint8_t reg, uint8_t *val, uint8_t antVal)...
void reset(void)...
uint8_t GetReg(uint8_t reg)...

/****************nrf-setup*************************/
void nrf24L01_init(void)...

/********************Reciver functions*****************************/
void receive_payload(void)...

/********************Transmitter functions*****************************/
void transmit_payload(uint8_t * W_buff)...

/********************Main*****************************/
int main(void)
{
    InitSPI();
    ioinit();
    INT0_interrupt_init();
    nrf24L01_init();

    SETBIT(PORTB,0);        //To se the LED is working an the chip is on
    _delay_ms(1000);
    CLEARBIT(PORTB,0);

    while(1)
    {
    }
    return 0;
}


ISR(INT0_vect)  //vektorn that is run when transmit_payload succeed or when receive_payload received data ...
```

If you are using the devise as a transmitter, I usually have an USART-interrupt function called "ISR(USART_RX_vect)" at the very bottom that triggers when the computer sends something to the microchip. In the usart interrupt vector it then calls transmit_payload with the data received from the usart.

If you don't use usart, in the main while loop, send the data as described above "Transmit data".

If you don't use the INT0-interrupt to check if the transmission succeeded put an if-statement in the main while loop to check whether the correct IRQ flag (nr 4) in the STATUS register is cleared as described earlier.

If your chip is in receiver mode, in the main while loop, i usually call:

```
while(1)
{
	reset();
	receive_payload();
}
```

And if i don't have an interrupt, followed by an if-statement to see if anything was received by checking if the correct IRQ flag (nr 6) is set!

Long range mode

If you have the + version, than you can set the RF_SETUP byte to 0x27 instead of 0x07, which will enable the 250 kbps mode (long range) on full power, and i also recommend you to read this tutorial on how to build an amplifying biquad antenna. (remember to set the EN_AA-delay to at least 500us as described above)

Registers

This is straight from the datashet of the older version of the nRF24L01 (i find it easier to read thean the + version)

This is the layout of the "top level" registers as i call them:

Instruction Name	Instruction Format [binary]	# Data Bytes	Operation
R_REGISTER	000A AAAA	1 to 5 LSByte first	Read registers. AAAAA = 5 bit Memory Map Address
W_REGISTER	001A AAAA	1 to 5 LSByte first	Write registers. AAAAA = 5 bit Memory Map Address *Executable in power down or standby modes only.*
R_RX_PAYLOAD	0110 0001	1 to 32 LSByte first	Read RX-payload: 1 – 32 bytes. A read operation will always start at byte 0. Payload will be deleted from FIFO after it is read. Used in RX mode.
W_TX_PAYLOAD	1010 0000	1 to 32 LSByte first	Used in TX mode. Write TX-payload: 1 – 32 bytes. A write operation will always start at byte 0.
FLUSH_TX	1110 0001	0	Flush TX FIFO, used in TX mode
FLUSH_RX	1110 0010	0	Flush RX FIFO, used in RX mode Should not be executed during transmission of acknowledge, i.e. acknowledge package will not be completed.
REUSE_TX_PL	1110 0011	0	Used for a PTX device Reuse last sent payload. Packets will be repeatedly resent as long as CE is high. TX payload reuse is active until W_TX_PAYLOAD or FLUSH TX is executed. TX payload reuse must not be activated or deactivated during package transmission
NOP	1111 1111	0	No Operation. Might be used to read the STATUS register

And here are all the other registers and there configurations:

Address (Hex)	Mnemonic	Bit	Reset Value	Type	Description
00	CONFIG				Configuration Register
	Reserved	7	0	R/W	Only '0' allowed
	MASK_RX_DR	6	0	R/W	Mask interrupt caused by RX_DR 1: Interrupt not reflected on the IRQ pin 0: Reflect RX_DR as active low interrupt on the IRQ pin
	MASK_TX_DS	5	0	R/W	Mask interrupt caused by TX_DS 1: Interrupt not reflected on the IRQ pin 0: Reflect TX_DS as active low interrupt on the IRQ pin
	MASK_MAX_RT	4	0	R/W	Mask interrupt caused by MAX_RT 1: Interrupt not reflected on the IRQ pin 0: Reflect MAX_RT as active low interrupt on the IRQ pin
	EN_CRC	3	1	R/W	Enable CRC. Forced high if one of the bits in the EN_AA is high
	CRCO	2	0	R/W	CRC encoding scheme '0' - 1 byte '1' – 2 bytes
	PWR_UP	1	0	R/W	1: POWER UP, 0:POWER DOWN
	PRIM_RX	0	0	R/W	1: PRX, 0: PTX
01	EN_AA Enhanced ShockBurst™				Enable 'Auto Acknowledgment' Function Disable this functionality to be compatible with nRF2401, see page 26
	Reserved	7:6	00	R/W	Only '00' allowed
	ENAA_P5	5	1	R/W	Enable auto ack. data pipe 5
	ENAA_P4	4	1	R/W	Enable auto ack. data pipe 4
	ENAA_P3	3	1	R/W	Enable auto ack. data pipe 3
	ENAA_P2	2	1	R/W	Enable auto ack. data pipe 2
	ENAA_P1	1	1	R/W	Enable auto ack. data pipe 1
	ENAA_P0	0	1	R/W	Enable auto ack. data pipe 0
02	EN_RXADDR				Enabled RX Addresses
	Reserved	7:6	00	R/W	Only '00' allowed
	ERX_P5	5	0	R/W	Enable data pipe 5.
	ERX_P4	4	0	R/W	Enable data pipe 4.
	ERX_P3	3	0	R/W	Enable data pipe 3.
	ERX_P2	2	0	R/W	Enable data pipe 2.
	ERX_P1	1	1	R/W	Enable data pipe 1.
	ERX_P0	0	1	R/W	Enable data pipe 0.
03	SETUP_AW				Setup of Address Widths (common for all data pipes)
	Reserved	7:2	000000	R/W	Only '000000' allowed
	AW	1:0	11	R/W	RX/TX Address field width '00' - Illegal '01' - 3 bytes '10' - 4 bytes '11' – 5 bytes LSByte will be used if address width below 5 bytes
04	SETUP_RETR				Setup of Automatic Retransmission
	ARD	7:4	0000	R/W	Auto Re-transmit Delay '0000' – Wait 250+86uS

Address (Hex)	Mnemonic	Bit	Reset Value	Type	Description
					'0001' – Wait 500+86uS '0010' – Wait 750+86uS …….. '1111' – Wait 4000+86uS (Delay defined from end of transmission to start of next transmission)[14]
	ARC	3:0	0011	R/W	Auto Retransmit Count '0000' –Re-Transmit disabled '0001' – Up to 1 Re-Transmit on fail of AA …… '1111' – Up to 15 Re-Transmit on fail of AA
05	RF_CH				RF Channel
	Reserved	7	0	R/W	Only '0' allowed
	RF_CH	6:0	0000010	R/W	Sets the frequency channel nRF24L01 operates on
06	RF_SETUP				RF Setup Register
	Reserved	7:5	000	R/W	Only '000' allowed
	PLL_LOCK	4	0	R/W	Force PLL lock signal. Only used in test
	RF_DR	3	1	R/W	Data Rate '0' – 1 Mbps '1' – 2 Mbps
	RF_PWR	2:1	11	R/W	Set RF output power in TX mode '00' – -18 dBm '01' – -12 dBm '10' – -6 dBm '11' – 0 dBm
	LNA_HCURR	0	1	R/W	Setup LNA gain
07	STATUS				Status Register (In parallel to the SPI instruction word applied on the MOSI pin, the STATUS register is shifted serially out on the MISO pin)
	Reserved	7	0	R/W	Only '0' allowed
	RX_DR	6	0	R/W	Data Ready RX FIFO interrupt. Set high when new data arrives RX FIFO[15]. Write 1 to clear bit.
	TX_DS	5	0	R/W	Data Sent TX FIFO interrupt. Set high when packet sent on TX. If AUTO_ACK is activated, this bit will be set high only when ACK is received. Write 1 to clear bit.
	MAX_RT	4	0	R/W	Maximum number of TX retries interrupt Write 1 to clear bit. If MAX_RT is set it must be cleared to enable further communication.
	RX_P_NO	3:1	111	R	Data pipe number for the payload

Address (Hex)	Mnemonic	Bit	Reset Value	Type	Description
					available for reading from RX_FIFO 000-101: Data Pipe Number 110: Not Used 111: RX FIFO Empty
	TX_FULL	0	0	R	TX FIFO full flag. 1: TX FIFO full. 0: Available locations in TX FIFO.
08	OBSERVE_TX				Transmit observe register
	PLOS_CNT	7:4	0	R	Count lost packets. The counter is overflow protected to 15, and discontinue at max until reset. The counter is reset by writing to RF_CH. See page 14 and 17.
	ARC_CNT	3:0	0	R	Count resent packets. The counter is reset when transmission of a new packet starts. See page 14.
09	CD				
	Reserved	7:1	000000	R	
	CD	0	0	R	Carrier Detect. See page 17.
0A	RX_ADDR_P0	39:0	0xE7E7E7E7E7	R/W	Receive address data pipe 0. 5 Bytes maximum length. (LSByte is written first. Write the number of bytes defined by SETUP_AW)
0B	RX_ADDR_P1	39:0	0xC2C2C2C2C2	R/W	Receive address data pipe 1. 5 Bytes maximum length. (LSByte is written first. Write the number of bytes defined by SETUP_AW)
0C	RX_ADDR_P2	7:0	0xC3	R/W	Receive address data pipe 2. Only LSB. MSBytes will be equal to RX_ADDR_P1[39:8]
0D	RX_ADDR_P3	7:0	0xC4	R/W	Receive address data pipe 3. Only LSB. MSBytes will be equal to RX_ADDR_P1[39:8]
0E	RX_ADDR_P4	7:0	0xC5	R/W	Receive address data pipe 4. Only LSB. MSBytes will be equal to RX_ADDR_P1[39:8]
0F	RX_ADDR_P5	7:0	0xC6	R/W	Receive address data pipe 5. Only LSB. MSBytes will be equal to RX_ADDR_P1[39:8]
10	TX_ADDR	39:0	0xE7E7E7E7E7	R/W	Transmit address. Used for a PTX device only. (LSByte is written first) Set RX_ADDR_P0 equal to this address to handle automatic acknowledge if this is a PTX device with Enhanced ShockBurst™ enabled. See page 14.
11	RX_PW_P0				
	Reserved	7:6	00	R/W	Only '00' allowed
	RX_PW_P0	5:0	0	R/W	Number of bytes in RX payload in data pipe 0 (1 to 32 bytes). 0 Pipe not used 1 = 1 byte … 32 = 32 bytes
12	RX_PW_P1				
	Reserved	7:6	00	R/W	Only '00' allowed
	RX_PW_P1	5:0	0	R/W	Number of bytes in RX payload in data pipe 1 (1 to 32 bytes). 0 Pipe not used

I hope you enjoyed my tutorial...
as always, if there is questions there might be an answer in the comment field.
/Kalle

Karl Hagström at 7:24 PM

Share G+

250 comments:

Andreas Kristensson April 10, 2013 at 10:00 AM

Kalle, great tutorials! Will use a Raspberry to control my sun shades (and then Nexa switches, etc) - the information you provide on your blog is spot on. Thanks

Reply